Homework

Basic:

1. 实现Phong光照模型：
   - 场景中绘制一个cube
   - **自己写shader**实现两种shading: Phong Shading 和 Gouraud Shading，并解释两种shading的实现原理
   - 合理设置视点、光照位置、光照颜色等参数，使光照效果明显显示
2. 使用GUI，使参数可调节，效果实时更改：
   - GUI里可以切换两种shading
   - 使用如进度条这样的控件，使ambient因子、diffuse因子、specular因子、反光度等参数可调节，光照效果实时更改

Bonus:

当前光源为静止状态，尝试使光源在场景中来回移动，光照效果实时更改。

1. 实现 **phong** 光照模型

   phong 光照模型顶点着色器 **PhongShader.v**
   变量 **pos** 代表顶点**位置**
   normal 代表顶点对应的**法向量**

   输出变量 **Normal** 为经 **model** 变换后的法向量；注意这一不能将传入的法向量直接左乘 **model** 矩阵得到变换后的法向量，因为变换不一定能**维持**法向量的一致性。

   需要将 **model 求逆并转置**后再乘以 normal 得到变换后的法向量。

   变量 **FragPos** 表示经 **model** 矩阵变换后的顶点位置。

```
#version 330 core
layout (location = 0) in vec3 pos;
layout (location = 1) in vec3 normal;

out vec3 Normal;
out vec3 FragPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(pos, 1.0f);
    FragPos = vec3(model * vec4(pos, 1.0));
    Normal = mat3(transpose(inverse(model))) * normal;
}
```

片段着色器 **PhongShader.f**

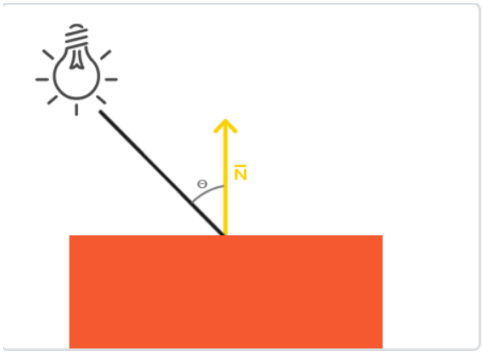
这里的输入变量为顶点着色器中计算好的 **Normal** 和 **FragPos** 变量;
由于变量从顶点着色器传入了片段着色器，每一点的位置和对应的法向量位置都已经根据顶点插值得到。所以传入的 **Normal** 和 **FragPos** 是每一个点根据顶点**插值**得到的位置和法向量信息。

此外这里还有四个 **uniform** 型的变量分别为物体的颜色 **objectColor**、光源的颜色 **lightColor**、视点位置 **viewPos** 和光源的位置 **lightPos。**

Phong 局部光照模型包括三个部分，分别为**环境光**、**漫反射**和**镜面反射**。

环境光：我们使用一个很小的常量（光照）颜色，添加到物体片段的最终颜色中，这样子的话即便场景中没有直接的光源也能看起来存在有一些发散的光。我们用**光的颜色**乘以一个很小的**常量环境因子**，再乘以**物体的颜色**，然后将最终结果作为片段的颜色：

漫反射：图左上方有一个光源，它所发出的光线落在物体的一个片段上。我们需要测量这个光线是以什么**角度**接触到这个片段的。如果光线垂直于物体表面，这束光对物体的影响会最大化。为了测量光线和片段的角度，我们使用法向量(Normal Vector)，它是垂直于片段表面的一个向量 (这里以黄色箭头表示)。我们知道两个单位向量的夹角越小，它们点乘的结果越倾向于 1。当两个向量的夹角为 90 度的时候，点乘会变为 0。这同样适用于 θ，θ 越大，光对片段颜色的影响就应该越小。

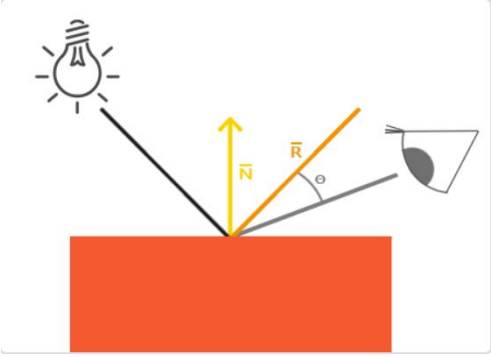


镜面反射：

和漫反射光照一样，镜面光照也是依据光的方向向量和物体的法向量来决定的，但是它也依赖于**观察方向**，例如玩家是从什么方向看着这个片段的。镜面光照是基于光的反射特性。如果我们想象物体表面像一面镜子一样，那么，无论我们从哪里去看那个表面所反射的光，镜面光照都会达到最大化。你可以从下面的图片看到效果：

我们通过反射法向量周围光的方向来计算反射向量。然后我们计算反射向量和视线方向的**角度差**，如果夹角越小，那么镜面光的影响就会越大。它的作用效果就是，当我们去看光被物体所反射的那个方向的时候，我们会看到一个高光。

观察向量是镜面光照附加的一个变量，我们可以使用观察者世界空间位置和片段的位置来计算它。之后，我们计算镜面光强度，用它乘以光源的颜色，再将它加上环境光和漫反射分量。




```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

in vec3 Normal;
in vec3 FragPos;

uniform vec3 viewPos;
uniform vec3 lightPos;

uniform float ambientStrength = 0.1;
uniform float specularStrength = 1;
uniform float specularFactor = 32;

void main()
{
    vec3 ambient = ambientStrength * lightColor;
```

```
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);

    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
specularFactor);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);

}
```

绘制一个 **cube**

立方体的顶点和对应的法向量的数据如下。

建立**立方体**和**光源**的 **VAO**,所用数据均为下方的 **vertices** 数组中的数据，不同的是在渲染**光源立方体**的时候**不需要**用到**法向量**的信息。

```
float vertices[] = {
    -0.2f, -0.2f, -0.2f, 1.0f, 1.0f, 1.0f,
    0.2f, -0.2f, -0.2f,0.0f, 1.0f, 1.0f,
    0.2f, 0.2f, -0.2f,0.0f, 0.0f, 1.0f,
    0.2f, 0.2f, -0.2f,0.0f, 0.0f, 1.0f,
    -0.2f, 0.2f, -0.2f,1.0f, 0.0f, 1.0f,
    -0.2f, -0.2f, -0.2f,1.0f, 1.0f, 1.0f,

    -0.2f, -0.2f, 0.2f,1.0f, 1.0f, 0.0f,
    0.2f, -0.2f, 0.2f,0.0f, 1.0f, 0.0f,
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,
    -0.2f, 0.2f, 0.2f,1.0f, 0.0f, 0.0f,
    -0.2f, -0.2f, 0.2f,1.0f, 1.0f, 0.0f,

    -0.2f, 0.2f, 0.2f,1.0f, 0.0f, 0.0f,
    -0.2f, 0.2f, -0.2f,1.0f, 0.0f, 1.0f,
    -0.2f, -0.2f, -0.2f,1.0f, 1.0f, 1.0f,
    -0.2f, -0.2f, -0.2f,1.0f, 1.0f, 1.0f,
    -0.2f, -0.2f, 0.2f,1.0f, 1.0f, 0.0f,
    -0.2f, 0.2f, 0.2f,1.0f, 0.0f, 0.0f,
```

```
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,
    0.2f, 0.2f, -0.2f,0.0f, 0.0f, 1.0f,
    0.2f, -0.2f, -0.2f,0.0f, 1.0f, 1.0f,
    0.2f, -0.2f, -0.2f,0.0f, 1.0f, 1.0f,
    0.2f, -0.2f, 0.2f,0.0f, 1.0f, 0.0f,
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,

    -0.2f, -0.2f, -0.2f,1.0f, 1.0f, 1.0f,
    0.2f, -0.2f, -0.2f,0.0f, 1.0f, 1.0f,
    0.2f, -0.2f, 0.2f,0.0f, 1.0f, 0.0f,
    0.2f, -0.2f, 0.2f,0.0f, 1.0f, 0.0f,
    -0.2f, -0.2f, 0.2f,1.0f, 1.0f, 0.0f,
    -0.2f, -0.2f, -0.2f,1.0f, 1.0f, 1.0f,

    -0.2f, 0.2f, -0.2f,1.0f, 0.0f, 1.0f,
    0.2f, 0.2f, -0.2f,0.0f, 0.0f, 1.0f,
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,
    0.2f, 0.2f, 0.2f,0.0f, 0.0f, 0.0f,
    -0.2f, 0.2f, 0.2f,1.0f, 0.0f, 0.0f,
    -0.2f, 0.2f, -0.2f,1.0f, 0.0f, 1.0f
};

unsigned int VBO, cubeVAO;
    glGenVertexArrays(1, &cubeVAO);
    glGenBuffers(1, &VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    glBindVertexArray(cubeVAO);


    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 *
sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    unsigned int lightVAO;
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
```

使用 **ImGui** 添加 **radioButton**，选择 **shader** 的类型；这里可以选择 phong 和 gouraud 两种类型。

```
ImGui::Text("Shading");
          ImGui::RadioButton("Phong Shading", &shaderMode,
PHONG);
          ImGui::RadioButton("Gouraud Shading", &shaderMode,
GOURAUD);
```

光源的位置是一个全局变量 `glm::vec3 lightPos(1.2f, 1.0f, 2.0f);`

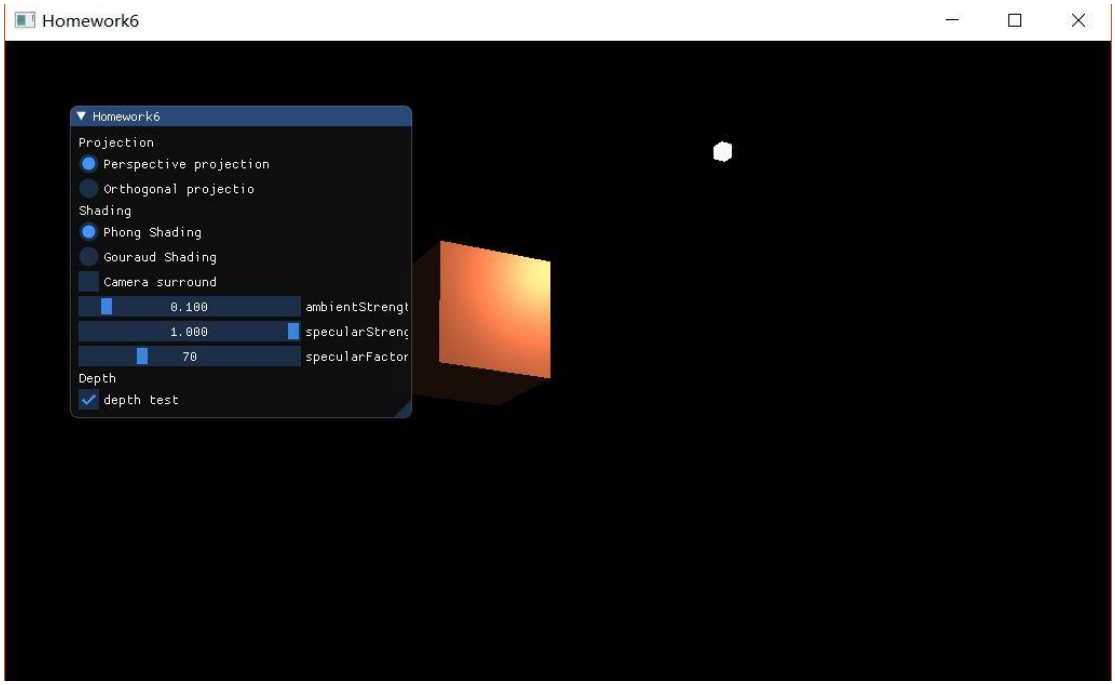
确定好这些之后，开始绘制**立方体**与**光源**

下面是选用 **phong 光照模型**时需要指定的参数。确定好之后使用 `glDrawArray` 函数绘制对应立方体即可。

```
glm::mat4 model = glm::mat4(1.0f);
glm::mat4 view = glm::mat4(1.0f);
glm::mat4 proj = glm::mat4(1.0f);
view = camera.GetViewMatrix();
radian = camera.Zoom;
if (shaderMode == PHONG) {
       phongLighting.useProgram();
       phongLighting.setModel(model);
       phongLighting.setView(view);
       phongLighting.setProjection(proj);

       phongLighting.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
       phongLighting.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
        phongLighting.setVec3("lightPos", lightPos.x, lightPos.y,
    lightPos.z);
       phongLighting.setVec3("viewPos", camera.Position.x,
camera.Position.y, camera.Position.z);
       phongLighting.setFloat("ambientStrength", ambientStrength);
          phongLighting.setFloat("specularStrength",
specularStrength);
       phongLighting.setInteger("specularFactor", specularFactor);
   }
```

```
glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    lampShader.useProgram();

    model = glm::translate(model, lightPos);
    model = glm::scale(model, glm::vec3(0.1f));

    lampShader.setModel(model);
    lampShader.setView(view);
    lampShader.setProjection(proj);

    glBindVertexArray(lightVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
```

效果：



Gouraud 光照模型

在实现 **Gouraud 光照模型**时，**不需要**在片段着色器中根据**每一点**的法向量单独的计算出每一点的颜色。而是在**顶点着色器**中将**顶点的颜色**计算好之后传入片段着色器，每点的颜色直接由**顶点颜色插值**得到即可。着色器代码如下：

GouraudShader.v

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
```

```
out vec3 LightingColor;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform float ambientStrength = 0.1;
uniform float specularStrength = 1;
uniform int specularFactor = 32;
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    vec3 Position = vec3(model * vec4(aPos, 1.0));
    vec3 Normal = mat3(transpose(inverse(model))) * aNormal;


    vec3 ambient = ambientStrength * lightColor;

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - Position);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;


    vec3 viewDir = normalize(viewPos - Position);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
specularFactor);
    vec3 specular = specularStrength * spec * lightColor;

    LightingColor = ambient + diffuse + specular;
}
```

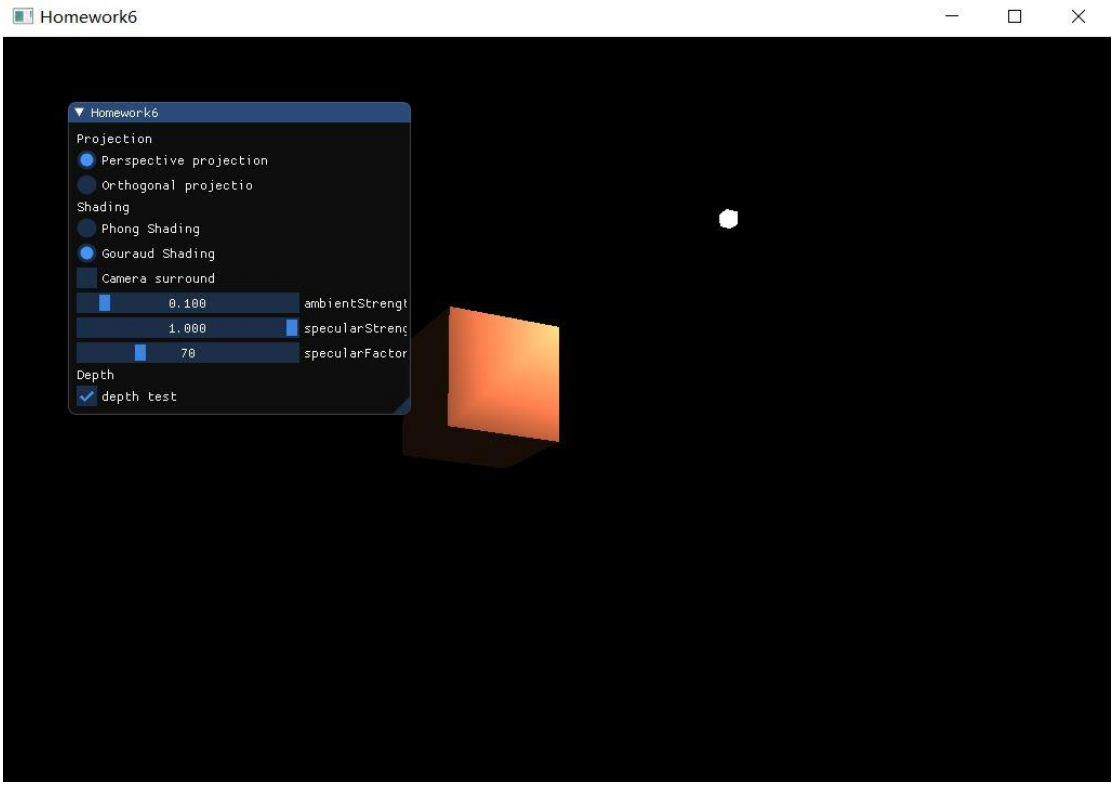
GouraudShader.f

```
#version 330 core
out vec4 FragColor;

in vec3 LightingColor;
```

```
uniform vec3 objectColor;

void main()
{
   FragColor = vec4(LightingColor * objectColor, 1.0);
}
```

LightingColor 变量即是根据顶点计算出的光照的颜色，传入片段着色器中后直接乘以物体颜色即可得到每一个像素点的颜色。

效果：



从图中可以明显的看到，使用 Gouraud 光照模型时由于没一点的颜色值是根据顶点的插值得到的，所以图中会有很明显的**光带**（正方形的左下到右上的对角线）。Phong 光照模型则效果要好的多，在**反射的方向**可以明显地看到反射产生的**圆形光斑**。

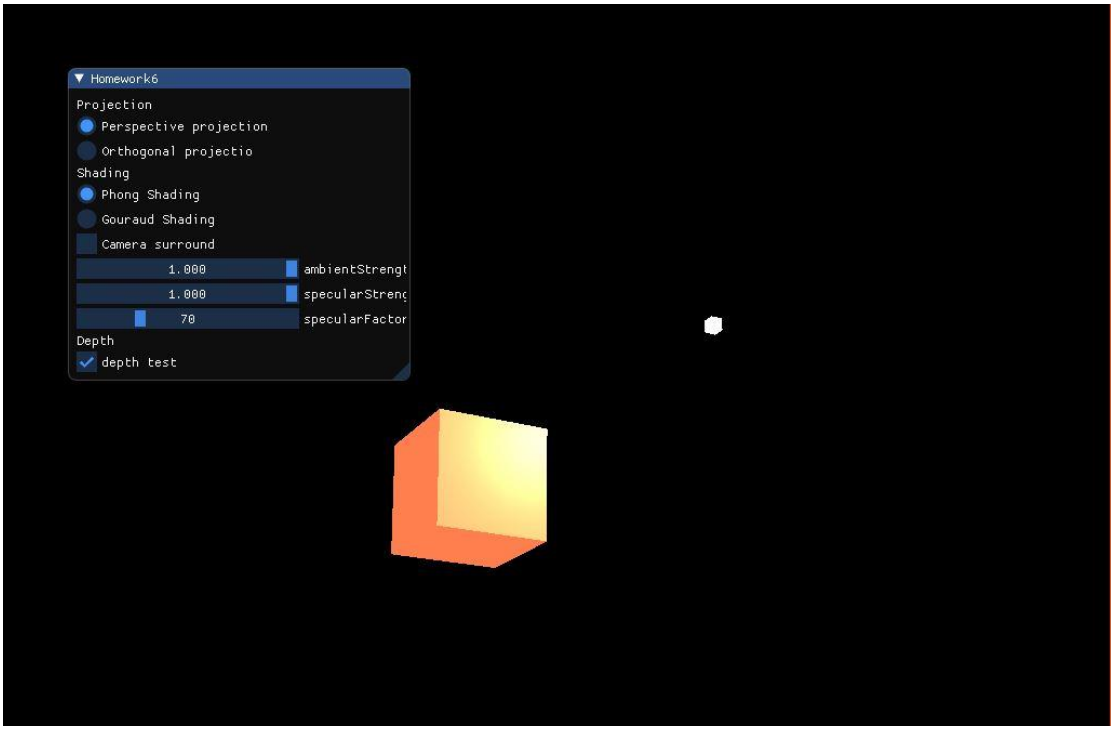
2. 使用 **GUI** 调节参数
   这里设置可以调节的参数为环境光的强度银子；镜面反射的强度银子与镜面反射的幂大小。

```
ImGui::SliderFloat("ambientStrength", &ambientStrength, 0.0, 1.0);
            ImGui::SliderFloat("specularStrength", &specularStrength, 0.0, 1.0);
```

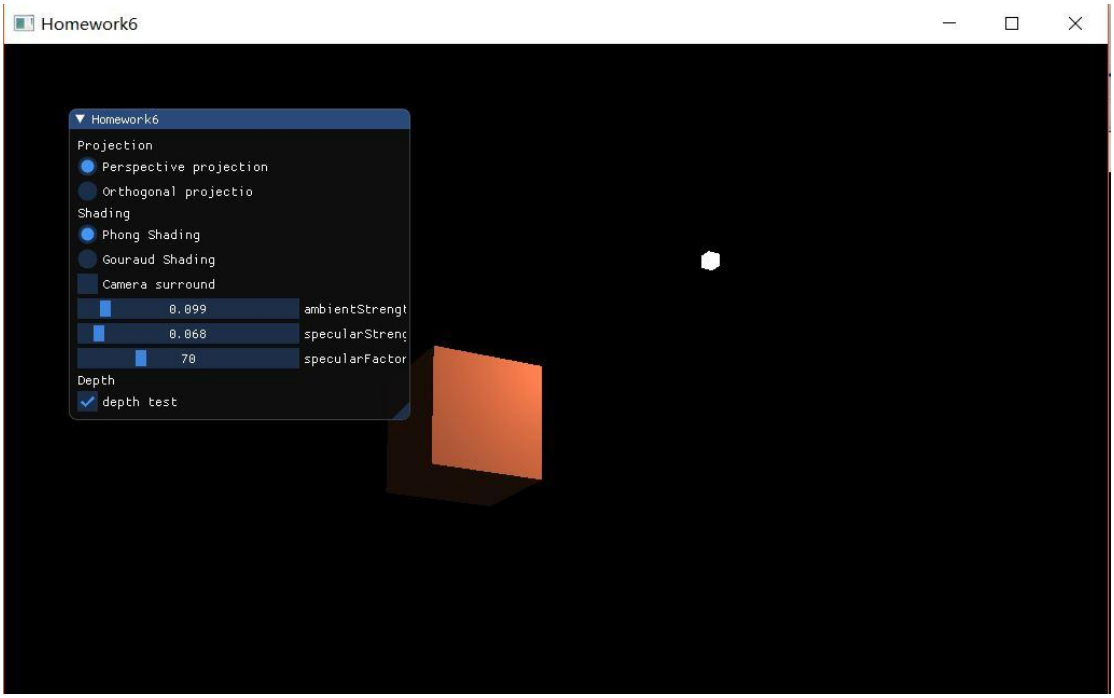
```
            ImGui::SliderInt("specularFactor", &specularFactor, 1,
256);
```

通过调节这些参数可以对光照模型进行改变。

增强**环境光**的强度:



减小**镜面反射**的强度:



3. 移动光源实时显示光照效果

通过 checkBox 选择是否选用移动光源(作业中截图时 text 有误, 这里应为 Light surround 而不是 Camera surrond)

```
ImGui::Checkbox("Light surround", &isLightSurround);
```

通过改变光源的位置使起运动轨迹为一个**椭圆**。效果在视频中展示。

```
if (isLightSurround) {
        lightPos.x = 2*sin(glfwGetTime());
        lightPos.y = cos(glfwGetTime());
        lightPos.z = 1;
}
```